8 Hashing: Open addressing

Summer Term 2011

Jan-Georg Smaus
Hashing: General Framework

Set of keys $S$

Universe $U$ of all possible keys

hash function $h$

hash table $T$

$h(s) = \text{hash address}$

$h(s) = h(s') \iff s \text{ and } s' \text{ are synonyms with respect to } h$;

address collision
Possible ways of treating collisions

- Collisions are treated differently in different methods.

- A data set with key $s$ is called a **colliding element** if bucket $B_{h(s)}$ is already taken by another data set.

- What can we do with colliding elements?
  1. **Chaining**: We learned about that in the last chapter.
  2. **Open Addressing**: Colliding elements are stored in other vacant buckets. During storage and lookup, these are found through so-called **probing**.
Open addressing

Idea:
Store colliding elements in vacant ("open") buckets of the hash table
If \(T[h(k)]\) is taken, find a different bucket for \(k\) according to a fixed rule

Example:
Consider the bucket with the next smaller index:
\[
(h(k) - 1) \mod m
\]

General:
Consider the sequence
\[
(h(k) - j) \mod m
\]
\(j = 0, \ldots, m-1\)
Probe sequence

Even more general:
Consider the probe sequence
\[(h(k) - s(j,k)) \mod m\]
\[j = 0, ..., m-1, \text{ for a given function } s(j,k)\]

Examples for the function
\[s(j, k) = j\] linear probing
\[s(j, k) = (-1)^j \* \left\lfloor \frac{j}{2} \right\rfloor^2\] quadratic probing

We will now look at these …

Others: uniform probing, random probing, double hashing
Probe sequence

Properties of $s(j,k)$

Sequence

$$(h(k) - s(0,k)) \mod m,$$
$$(h(k) - s(1,k)) \mod m,$$

$$(h(k) - s(m-2,k)) \mod m,$$
$$(h(k) - s(m-1,k)) \mod m$$

should result in a permutation of $0, \ldots, m-1$.

Example: Quadratic probing

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
</table>

$h(11) = 4$

Critical:

$s(j,k) = -1,1,-4,4,-9,9$

Deletion of data sets $\rightarrow$ mark as deleted

(Insert 4, 18, 25; delete 4; lookup 18, 25)
class OpenHashTable extends HashTable {
    // in HashTable: TableEntry [] T;
    private int [] tag;
    static final int EMPTY = 0;
    static final int OCCUPIED = 1;
    static final int DELETED = 2;
    // Constructor
    OpenHashTable (int capacity) {
        super(capacity);
        tag = new int [capacity];
        for (int i = 0; i < capacity; i++) {
            tag[i] = EMPTY;
        }
    }
    // The hash function
    protected int h (Object key) {...}
    // Function s for probe sequence
    protected int s (int j, Object key) {
        // quadratic probing
        if (j % 2 == 0)
            return ((j + 1) / 2) * ((j + 1) / 2);
        else
            return -((j + 1) / 2) * ((j + 1) / 2);
    }
public int searchIndex (Object key) {
    /* searches for an entry with the given key in the hash table and
     * returns the respective index or -1 */
    int i = h(key);
    int j = 1; // next index of probing sequence
    while (tag[i] != EMPTY && !key.equals(T[i].key)){
        // Next entry in probing sequence
        i = (h(key) - s(j++, key)) % capacity;
        if (i < 0)
            i = i + capacity;
    }
    if (key.equals(T[i].key) && tag[i] == OCCUPIED)
        return i;
    else
        return -1;
}

public Object search (Object key) {
    /* searches for an entry with the given key in the hash table and
     * returns the respective value or NULL */
    int i = searchIndex (key);
    if (i >= 0)
        return T[i].value;
    else
        return null;
}
public void insert (Object key, Object value) {
    // inserts an entry with the given key and value
    int j = 1;  // next index of probing sequence
    int i = h(key);
    while (tag[i] == OCCUPIED) {
        i = (h(key) - s(j++, key)) % capacity;
        if (i < 0)
            i = i + capacity;
    }
    T[i] = new TableEntry(key, value);
    tag[i] = OCCUPIED;
}
public void delete (Object key) {
    // deletes entry with given key from the hash table
    int i = searchIndex(key);
    if (i >= 0) {
        // Successful search
        tag[i] = DELETED;
    }
}
Test program

```java
public class OpenHashingTest {
    public static void main(String args[]) {
        Integer[] t = new Integer[args.length];
        for (int i = 0; i < args.length; i++)
            t[i] = Integer.valueOf(args[i]);
        OpenHashTable h = new OpenHashTable(7);
        for (int i = 0; i <= t.length - 1; i++) {
            h.insert(t[i], null);
            if (i != t.length - 1) {  // added newline for readability
                h.printTable();
            }
        }
        h.delete(t[0]); h.delete(t[1]);
        h.delete(t[6]); h.printTable();
    }
}

Call:
    java OpenHashingTest 12 53 5 15 2 19 43

Output (quadratic probing):
[ ] [ ] [ ] [ ] [ ] [ ] (12) [ ]
[ ] [ ] [ ] [ ] (53) (12) [ ]
[ ] [ ] [ ] [ ] (53) (12) (5)
[ ] (15) [ ] [ ] (53) (12) (5)
(19) (15) (2) [ ] (53) (12) (5)
(19) (15) (2) (43) (53) (12) (5)
(19) (15) (2) {43} {53} {12} (5)
```
Probe sequences – linear probing

\[ s(j,k) = j \]

Probe sequence for \( k \):

\[ h(k), h(k)-1, \ldots, 0, m-1, \ldots, h(k)+1, \]

Problem:

“primary clustering”

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
\hline
\text{5} & \text{53} & \text{12} & & & \\
\end{array}
\]

\[ Pr \text{ (next object ends at position 2) } = 4/7 \]

\[ Pr \text{ (next object ends at position 1) } = 1/7 \]

Long chains are extended with higher probability than short ones.
Efficiency of linear probing

Successful search:

\[ C_n \approx \frac{1}{2} \left( 1 + \frac{1}{1-\alpha} \right) \]

Failed search:

\[ C'_n \approx \frac{1}{2} \left( 1 + \frac{1}{(1-\alpha)^2} \right) \]

<table>
<thead>
<tr>
<th>( \alpha )</th>
<th>( C_n ) (successful)</th>
<th>( C'_n ) (failed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.50</td>
<td>1.5</td>
<td>2.5</td>
</tr>
<tr>
<td>0.90</td>
<td>5.5</td>
<td>50.5</td>
</tr>
<tr>
<td>0.95</td>
<td>10.5</td>
<td>200.5</td>
</tr>
<tr>
<td>1.00</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Efficiency of linear probing decreases drastically as soon as the load factor \( \alpha \) gets close to the value 1.
Quadratic probing

\[ s(j,k) = (-1)^j * \left[ \frac{j}{2} \right]^2 \]

Probe sequence for \( k \):

\[ h(k), \ h(k)+1, \ h(k)-1, \ h(k)+4, \ ... \]

Permutation, if \( m \) is a prime of the form \( 4i + 3 \), for some \( i \).

**Problem**: secondary clustering, i.e. two synonyms \( k \) and \( k' \) always run through the same probe sequence.
Efficiency of quadratic probing

Successful search:

\[ C_n \approx 1 - \frac{\alpha}{2} + \ln \left( \frac{1}{1-\alpha} \right) \]

Failed search:

\[ C'_n \approx \frac{1}{1-\alpha} - \alpha + \ln \left( \frac{1}{1-\alpha} \right) \]

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$C_n$ (successful)</th>
<th>$C'_n$ (failed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.50</td>
<td>1.44</td>
<td>2.19</td>
</tr>
<tr>
<td>0.90</td>
<td>2.85</td>
<td>11.40</td>
</tr>
<tr>
<td>0.95</td>
<td>3.52</td>
<td>22.05</td>
</tr>
<tr>
<td>1.00</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>